# ASSIGNMENT 3 REPORT

COSC2769 – Further Web Development

Group 15

# Contents

# I. Introduction

For this assignment, it was given that teams should create an ecommerce website based on the requirements. This report will detail the process, design choices, and implementation of features.

# II. Team Member

Le Viet Bao - s3979654

- Viet Bao is a first-year student, studying Information Technology as a major. Even though he is relatively inexperienced, Bao has played some major roles within past projects, including being a project manager, in charge of creating full stack web applications as an assessment.

To Bao Minh Hoang - s39785554

- Minh Hoang is a first-year student, minoring in ‘Mobile and Web Development’, under the Bachelor of Information Technology program. His past accolades include developing an ecommerce website, and other various software and hardware related projects. Partaking in this project with the reason of bettering himself as a programmer, and team player.

Ngo Chi Binh - s3938145:

- Ngo Chi Binh is currently a second-year student majoring in ‘Information Technology’, with a minor in ‘Mobile and Web Development’. His past academic projects include web applications written in Java and JavaScript and Pythons. He has aspirations of becoming a Web Security Specialist and believes that learning how to develop web applications to be an important part of that goal.

Trinh Quang Minh - s3848088

- Quang Minh is currently a third-year student majoring in ‘App Development’. His past projects include a product scanning app for supermarkets, an accommodation booking app, and shopping website. He mostly works on the designing of the interface as well as working on the testing and reporting of the projects.

# III. Design

## 1. High-level Design

This section will delve into the application architecture foundations, key components, and their interactions to make up the final product.

### a. Application Architecture

Fundamentally, the application relies on a carefully considered three-tier architecture, serving as the scaffolding that holds the components together. It encompasses the following tiers:

Presentation Tier: The user interface was constructed using React.js , a dynamic Javascript library. React.js was chosen due to its component-based architecture, making scalability and integrations more of a simpler process. Not to mention, its wide variety of learning material that is available on the Internet, perfectly suitable for a team as such.

Application Logic Tier: To handle user requests, data processing, and bridging between the two ends, Node.js and Express.js, was chosen, not only for its reliability, ease of use, and efficient server-side handling, but for the team's overall familiarity with the tools.

Data Tier: MongoDB was chosen because of one simple factor, familiarity. Any web development projects the team have partaken up until now, has been using MongoDB as a means to store data. Additionally, its functionality is intuitive and the learning curve is not steep, with chances for scalability and modifications.

## b. Components and Their Interplay

Frontend: React components were used to render the user interface, with each component triggering a HTTP request to the Backend, initiating critical processes like data retrieval, order creation, and authentication.

Backend: Modules, created using Node.js and Express.js modules, were employed to handle requests, execute logic, and facilitate communication with the database. Furthermore, the modules are responsible for user authentication and authorization, ensuring proper security.

Database: The database component takes on a pivotal role of data management, serving as a repository for all of the necessary schemas and orders. Structurally important in maintaining the integrity and consistency of the application.

## c. Interaction Among Components

Frontend to Backend: When an action is initiated by the user from the user interface, it will set off a cascading event of HTTP requests towards the backend. In turn, libraries such as Axios, will facilitate the request for a myriad of purposes such as fetching products, order creation, etc.

Backend to Database: Since the application uses a nonSQL database, interactions between backend and database are defined within the functions, components. Thus, ensuring the accuracy and currency of information within the application.

Authentication: The Backend component serves as a means to authenticate users, validating the user based on existing schemas, when deemed appropriate, tokens will be issued as keys to access specific sessions of the application.

# 2. Low-Level Design

While the high-level design offers a panoramic view of our application, the low-level design embarks on a voyage of intricate specifics, delving deep into the internal workings of each major component. It unveils their structural intricacies, functionalities, and the meticulous design choices that underpin their implementation.

## a. Frontend

Component Structure: The application's React components are organised in a manner that promotes scalability, readability. With each carrying a specific purpose and functionality, furthermore, aiding with the process of maintenance. There are certainly many ways to structure the components, but the team was most comfortable with the current layout.

With styling, Bootstrap was enlisted for usage, due to its convenience and scalability, even if there was a member who was not used to Bootstrap, its intuitiveness and the number of available resources, made it a perfect candidate. Onto the application itself. The team decided on a minimalist aesthetic, boasting peak usability and functionality with little distracting visual noise. Further aiding intuitiveness, the product's layout is simple to grasp, with the navigation bar that is not clouded with menus. There were mentions of using external CSS, or other styling libraries, such as Tailwind. but the latter required additional learning and training as less members are used to it. The former, on the hand, is basically common knowledge, everyone can confidently use it, but due it being an external file, there are some additional routings that needed to be done, not to mention the fact that CSS lack the scalability and efficiency of Bootstrap.

Routing: The application greatly utilised Express's routing function, with the backend code consisting mostly of this feature. react-router-dom is also utilised, for a myriad of purposes such as authentication and loading the necessary content into the Frontend components. Though there are some improvements here and there, considering most of the main server-side handling consist within a singular server file, greatly dampening the process of troubleshooting and maintenance.

Database Interaction: The database interactions are facilitated by MongoDB. The library provides functions such as findbyID(), findOne(), which are integral to the general functionality of the application, as most of information regarding a specific object are stored under the guise of objectID, provided by MongoDB.

## b. Database

Schema Design: Overall, the schemas were throughout based on the team's needs and the required functionality. One of the more notable examples is our user Schema, instead of nesting the roles into a singular schema, they are separated into three different Schemas, Seller, Customer, and Warehouse Admin. The latter is leagues more complicated to implement compared to the former, but it paid off by not requiring the code to have various If() for different roles, streamlining the production process.

# IV. Implementation

## 1. Signup and login system

The application gives the user the ability to sign up for a new account, or log into an existing one, with sufficient security and authorization for both routes.

```
//=============Sign/Auth==========//

// Register a new user

app.post("/register", async (req, res) => {
  const { email,phone, password, role, businessName } = req.body;

  // Hash the password (you'll need to install bcrypt)
  const hashedPassword = bcrypt.hashSync(password, 10);

  let newUser;

  if (role === "Admin") {
    newUser = new WH_Admin({ email, password: hashedPassword, role });
  } else if (role === "Seller") {
    newUser = new Seller({ email, password: hashedPassword, role, phone, businessName });
  } else {
    newUser = new Customer({ email, password: hashedPassword, role, phone });
  }

  await newUser.save();
  res.status(201).json({ message: "User registered", newUser });
});
```

```
//=============Login/Auth============//

// Login
app.post("/login", async (req, res) => {
  const { email, password } = req.body;

  // Find user by email (you'll need to search in all collections)

  let user =
    (await WH_Admin.findOne({ email })) ||
    (await Seller.findOne({ email })) ||
    (await Customer.findOne({ email }));

  if (!user) {
    return res.status(401).json({ message: "Invalid credentials" });
  }

  // Validate password
  const validPassword = bcrypt.compareSync(password, user.password);

  if (!validPassword) {
    return res.status(401).json({ message: "Invalid credentials" });
  }

  // Generate JWT token
  const token = jwt.sign(
    { id: user._id, role: user.role },
    process.env.SECRET_KEY,
    {
      expiresIn: "1h",
    }
  );

  res.status(200).json({ token });
});

// Start the server
app.listen(port, () => {
  console.log(`Server running on http://localhost:${port}`);
});
```

# 2. Authentication

How users are authenticated within the system is quite interesting. As mentioned above, this method requires three separate Schemas for three user roles, that being Customer, Seller, and Warehouse Administrator. Here is how it function in layman's term:

- Upon logging in, the inputted data will be run through, based on which role the user chooses to log in as, the respective database. If the credentials are valid, the user will receive a JWT token that is stored in the browser's local storage, given that it will self-destruct after a certain amount of time. Based on that token, the user will be allowed to perform tasks based on the roles.

```
module.exports = passport => {
  passport.use(
    new JwtStrategy(opts, async (jwt_payload, done) => {
      try {
        let user = null;

        // Search in different collections based on role
        if (jwt_payload.role === 'Admin') {
          user = await WH_Admin.findById(jwt_payload.id);
        } else if (jwt_payload.role === 'Seller') {
          user = await Seller.findById(jwt_payload.id);
        } else if (jwt_payload.role === 'Customer') {
          user = await Customer.findById(jwt_payload.id);
        }

        if (user) {
          return done(null, user);
        }
        return done(null, false);
      } catch (err) {
        return done(err, false);
      }
    })
  );
};
```

```
// Create the authentication context
const AuthContext = createContext();

// Define the possible actions for the authReducer
const ACTIONS = {
  setToken: "setToken",
  clearToken: "clearToken",
};

// Reducer function to handle authentication state changes
const authReducer = (state, action) => {
  switch (action.type) {
    case ACTIONS.setToken:
      // Set the authentication token in axios headers and local storage
      axios.defaults.headers.common["Authorization"] = "Bearer " + action.payload;
      localStorage.setItem("token", action.payload);

      // Update the state with the new token
      return { ...state, token: action.payload };
```

- Another thing that the token can do is store data of the user, thus the reason that these middlewares exist:

```
app.get("/sellerOrders", getSellerFromJwt, async (req, res) => {
  const sellerId = req.seller._id;

  try {
    // Find orders containing products sold by the seller
    const orders = await Order.find({ "product.seller": sellerId })
      .populate('product.productId'); // Populate product details

    // Fetch additional customer details separately
    for(let order of orders) {
      order.customer = await Customer.findById(order.customer).select('email');
    }

    res.status(200).json(orders);
  } catch (error) {
    console.error(error); // Log the error to console for debugging
    res.status(500).json({ message: "Error fetching seller orders" });
  }
});
```

```
/// Middleware to get customer data from JWT payload
async function getCustomerFromJwt(req, res, next) {
  try {
    const authHeader = req.headers.authorization;

    if (!authHeader) {
      return res.status(401).json({ message: "No token provided" });
    }

    const token = authHeader.split(" ")[1]; // Extract token from Bearer

    // Decode the token without verifying it to inspect its payload
    const decoded = jwt.decode(token);

    console.log("Decoded JWT:", decoded);

    if (!decoded || !decoded.id) {
      return res.status(401).json({ message: "Invalid token" });
    }

    // Now we use the ID from the decoded JWT payload to find the customer
    const customer = await Customer.findById(decoded.id);

    if (!customer) {
      return res.status(404).json({ message: "Customer not found" });
    }

    req.customer = customer;
    console.log("Customer data from JWT:", req.customer);
    next();
  } catch (error) {
    res.status(500).json({ message: "Server Error" });
  }
}
```

# 3. Database security

To hash and salt user's password, the team employed an extension called bcrypt:

```
app.post("/register", async (req, res) => {
  const { email,phone, password, role, businessName } = req.body;

  // Hash the password (you'll need to install bcrypt)
  const hashedPassword = bcrypt.hashSync(password, 10);

  let newUser;

  if (role === "Admin") {
    newUser = new WH_Admin({ email, password: hashedPassword, role });
  } else if (role === "Seller") {
    newUser = new Seller({ email, password: hashedPassword, role, phone, businessName });
  } else {
    newUser = new Customer({ email, password: hashedPassword, role, phone });
  }

  await newUser.save();
  res.status(201).json({ message: "User registered", newUser });
});
```

```
"dependencies": {
  "bcrypt": "^5.1.1",
  "bcryptjs": "^2.4.3",
```

```
password: "$2b$10$YwUUJQ3rpSE/aQ1g6D2Rf.up6.IKPHcH6LhvkpdJus6NlZh131Yfa"
```

# 4. Order management system:

It starts at the customer, where they select the items that they want to buy, in turn, those items are stored into local storage of their browser. Upon entering the cart page, the said items will be displayed. When the customer pressed the "Place order" button, a post request will be sent to the server, creating the order right then. Customers can also see their order history, and which products have been "Cancelled" or "Shipped". When the product has the status of "Shipped", they are allowed to reject or accept the product.

```
// Customer can accept or reject products in their order
app.patch(
  "/updateProductCustomerStatus/:orderId/:productId/:customerStatus",
  getCustomerFromJwt,
  async (req, res) => {
    const { orderId, productId, customerStatus } = req.params;

    try {
      // Find the order
      const order = await Order.findById(orderId);
      if (!order) {
        return res.status(404).json({ message: "Order not found" });
      }

      // Check if the order belongs to the logged-in customer


      // Check if the order belongs to the logged-in customer
      if (!order.customer.equals(req.customer._id)) {
        return res.status(403).json({ message: "Not authorized to update this order" });
      }

      // Find the product in the order
      console.log('Order products:', order.product);

      const productInOrder = order.product.find((product) =>
      product._id.equals(productId)
    );


      if (!productInOrder) {
        return res
          .status(404)
          .json({ message: "Product not found in the order" });
      }

      if (productInOrder.sellerStatus !== "Shipped") {
        return res
          .status(400)
          .json({ message: 'Product status must be "Shipped" to update' });
      }
```

```
    if (customerStatus === "Accepted" || customerStatus === "Rejected") {
      productInOrder.customerStatus = customerStatus;
      await order.save();
      res
        .status(200)
        .json({ message: `Product status updated to "${customerStatus}"` });
    } else {
      res.status(400).json({ message: "Invalid status" });
    }
  } catch (error) {
    console.error(error);
    res.status(500).json({ message: "Error updating product status" });
  }
}
);
```

```
// Place an order
app.post("/placeOrder", getCustomerFromJwt, async (req, res) => {
  console.log("pp", req.customer);
  console.log("Request Body:", req.body);

  const { cart, totalPrice } = req.body;  // Extract totalPrice from req.body

  try {
    // Create a new order with initial status "New"
    const order = await Order.create({
      customer: req.customer._id,
      product: cart, // Use the "cart" array as the products in the order
      status: "New",
      totalPrice,  // Save the totalPrice to the database
    });

    console.log("Cart:", cart); // Log the cart for debugging
    console.log("Total Price:", totalPrice); // Log the total price for debugging

    console.log("Order created:", order); // Log the created order if it's successful

    res.status(201).json(order);
  } catch (error) {
    console.error("Error placing order:", error); // Log the error for debugging
    res
      .status(500)
      .json({ message: "Error placing order", error: error.message });
  }
});
```

On the seller's side, they can see the order that the customer has placed, though they can only see their own products within the order. Additionally, they can either change the status of the product to either “Cancelled” or “Shipped”. In turn, these statuses can be compiled and counted, turning it into a seller's statistic.

```
app.get("/sellerOrders", getSellerFromJwt, async (req, res) => {
  const sellerId = req.seller._id;

  try {
    // Find orders containing products sold by the seller
    const orders = await Order.find({ "product.seller": sellerId })
      .populate('product.productId'); // Populate product details

    // Filter the products in each order to only include products sold by the current seller
    orders.forEach(order => {
      order.product = order.product.filter(prod => prod.seller.equals(sellerId));
    });

    // Fetch additional customer details separately
    for(let order of orders) {
      order.customer = await Customer.findById(order.customer).select('email');
    }

    res.status(200).json(orders);
  } catch (error) {
    console.error(error); // Log the error to console for debugging
    res.status(500).json({ message: "Error fetching seller orders" });
  }
});
```

```
app.patch("/updateProductStatus/:orderId/:productId", getSellerFromJwt, async (req, res) => {
  const { orderId, productId } = req.params;
  const { sellerStatus } = req.body;
  const sellerId = req.seller._id;

  // Validate the sellerStatus value before proceeding
  if (!['Canceled', 'Shipped', 'Pending'].includes(sellerStatus)) {
    return res.status(400).json({ message: "Invalid seller status value" });
  }

  try {
    const order = await Order.findById(orderId);

    if (!order) {
      return res.status(404).json({ message: "Order not found" });
    }

    console.log("Order Products:", JSON.stringify(order.product, null, 2)); // Log to see the actual product details in the order
    console.log("Seller ID from token:", sellerId); // Log to see the actual seller ID from the token

    const productIndex = order.product.findIndex(
      (p) => p._id.toString() === productId && p.seller.toString() === sellerId.toString()
    );


    if (productIndex === -1) {
      return res.status(404).json({ message: "Product not found in order or you are not authorized to update this product" });
    }

    // Update the product status in the order
    order.product[productIndex].sellerStatus = sellerStatus;

    await order.save();
    res.status(200).json(order);
  } catch (error) {
    console.error(error);
    res.status(500).json({ message: "Error updating product status", error: error.message });
  }
});
```

```
app.get("/sales-statistics/", getSellerFromJwt, async (req, res) => {
  try {
    const sellerId = req.seller._id; // Adjusted to use req.seller._id to get the seller ID

    // Validate seller ID
    const seller = await Seller.findById(sellerId);
    if (!seller) {
      return res.status(404).json({ message: "Seller not found" });
    }
    console.log(seller);

    // Gather sales statistics
    const orders = await Order.find().populate({
      path: "product.productId",
      populate: {
        path: "seller",
        model: "Seller",
      },
    });

    const stats = {
      new: 0,
      shipped: 0,
      canceled: 0,
      accepted: 0,
      rejected: 0,
    };

    orders.forEach((order) => {
      order.product.forEach((product) => {
        // Check if the product belongs to the seller we are calculating the stats for
        if (product.seller.toString() === sellerId.toString()) {
          if (product.sellerStatus === "Shipped") {
            stats.shipped += 1;
          } else if (product.sellerStatus === "Canceled") {
            stats.canceled += 1;
          }

          if (product.customerStatus === "Accepted") {
            stats.accepted += 1;
          } else if (product.customerStatus === "Rejected") {
            stats.rejected += 1;
          } else if (product.customerStatus === "Pending") {
            stats.new += 1;
          }
        }
      });
    });
```

CRUD for Warehouse Administrators and seller approval system

- The CRUD allows them to create, read, update, and delete categories for products. This relies on the conventional request and response from both Backend and Frontend.

```
// Create a New Category
app.post("/addCategory", async (req, res) => {
  const { name, parent } = req.body;

  // Find the parent category by its name
  const parentCategory = await Category.findOne({ name: parent });

  if (parent && !parentCategory) {
    return res.status(404).json({ message: "Parent category not found" });
  }

  // Use the ObjectId of the parent category
  const newCategory = new Category({
    name,
    parent: parent ? parentCategory._id : null,
  });

  try {
    await newCategory.save();
    res.status(201).json(newCategory);
  } catch (error) {
    console.log("Error:", error);
    res.status(500).json({ message: "Internal Server Error" });
  }
});
```

- Furthermore, admins have the ability to approve new sellers, as upon creating a new seller account, the status of said seller account will be “Pending”. Upon approval by the admin, the status will be changed to “Approved”. This function and middleware will find and update sellers using their email, the reason for this is for ease of usage.

```
// Update a seller's status by Email
app.patch("/sellers/:email", getSellerByEmail, async (req, res) => {
  console.log(req.seller);
  try {
    const { status } = req.body;

    // Update seller status
    if (status && ["Approved", "Rejected"].includes(status)) {
      req.seller.status = status;
      await req.seller.save();
      res.json({ message: "Seller status updated successfully" });
    } else {
      res.status(400).json({ message: "Invalid status value" });
    }
  } catch (error) {
    res.status(400).json({ message: error.message });
  }
});

// Middleware to get a specific seller by Email
async function getSellerByEmail(req, res, next) {
  try {
    const seller = await Seller.findOne({ email: req.params.email });
    if (!seller) {
      return res.status(404).json({ message: "Seller not found" });
    }
    req.seller = seller;
    console.log(req.seller);
    next();
  } catch (error) {
    res.status(500).json({ message: "Internal Server Error" });
  }
}
```

# V. Testing

For the testing process, we had meetings with each other and compiled everything together to have a test run for every functionality that we had created for the websites. In order to fully test our project, we have tested through many different steps:

| | |
|---|---|
| Requirement Analysis | ● Check with the requirements we have from the lecturer as well as the Lazada's requirements<br>● Check if the final products have all the necessary requirements |
| Test Planning | ● Divide the time to test each section of the website<br>● Divide testing sections to each member<br>● Find the goals and outcomes of the whole testing process<br>● Prepare test data |
| Functional Testing | ● Test all interactive items (buttons, links, etc.)<br>● Test all data validation and handling users' input |
| Compatibility Testing | ● Test on different browsers (Chrome, Firefox, MS Edge, etc.)<br>● Test on different operating systems (Windows, MacOS, Linux) |
| Responsive Design Testing | ● Utilise different emulators and adjust the browser window's size to test the responsiveness of the website.<br>● Make sure the layout adapts to various screen widths without any issues. |
| Performance Testing | ● To evaluate the web page's performance under significant usage, undertake load testing.<br>● Calculate website load times and locate performance stumbling blocks.<br>● Improve load times by optimising code and assets.<br>● Test in various network environments (3G, 4G, and Wi-Fi) to assess performance variances. |
| Security Testing | ● To find potential threats, run security scans and vulnerability assessments.<br>● Check for security flaws like SQL injection, XSS, CSRF, and incorrect security configurations.<br>● Make careful to keep important data safely and with encryption. |
| Accessibility Testing | ● Make sure everyone can access and use the website, including those with disabilities.<br>● Addressing accessibility concerns and offering substitute content where needed |
| Usability Testing | ● To assess the usability of a website, engage actual users or usability testers (students and/or lecturers).<br>● Gather opinions on the design, navigation, and overall usability. |

| | |
|---|---|
| | ● Based on the outcomes of usability testing, make improvements. |
| Cross-browser Testing | ● Retest essential features to find compatibility problems brought on by code updates across various browsers and versions.<br>● To speed up cross-browser testing, use browser testing tools or services. |
| User Acceptance Testing (UAT) | ● Participate end users or stakeholders to conduct UAT in a controlled setting.<br>● Make sure the website satisfies both user and corporate objectives.<br>● Before the production release, fix any problems found during UAT. |

# VI. Known Issues and Missing Features

## 1. Known Issues

### a. Update categories for Admin

Every time there is an attempt at this function, it returns a server error message, the team has been sitting on this issue for days with no solutions. Ultimately, it was left behind as an issue.

## 2. Missing features

### a. Server-side cart

One of the requirements of the cart is for it to be able to be stored on the server if the user ever switches the browsers. Though there were attempts at this feature, it ultimately returns fruitless. The remnants of this attempt is the cart field in the customer schema.

### b. Filtering feature

The product is also missing a feature, that is the filter, that allows users to filter out prices. The exclusion of this feature was purely time constraint, as other functions were prioritised.

# VII. Development Process

This part will go over the development process of the application.

During the first phase of the project, which was mostly planning and discussion, the team got together and drafted a rough sketch of what the website will be, with members giving their own inputs, resulting in a long list of features and designs. Though chaotic, it was to be expected,

as later on, the team would go through the requirements and sort out the list, putting it in a hierarchical structure of importance. After which, the team would decide on workload, whose responsibility for each section, dividing the group into two parts, front-end and back-end. With the general goals established, the sub-groups would further divide the workload to more manageable chunks and keep track of a list, so the rest of the team were aware of the process. It is worth noting that the team roles did not include a leader, in the traditional sense, all members functioned as equals, with the occasional reminder to keep on track. This was unanimously agreed upon, for each member to have their own chance to exceed and gain experience.

For the second phase, our front-end team worked on a rough sketch of the website layout, giving everyone an idea of what the final product would look like. Whilst the back-end team started their work on the server side of things. This was set out to be a tedious process, due to the overall lack of experience within members, in turn, creating a notable time sink in learning the necessary skills in order to create the application. There were attempts to simplify the process by using AI assistance, though it was proven to be as time consuming as manual work.

After preparation was completed during the second phase, the third phase marked the beginning of the treacherous process of combining and debugging the code. This phase would consist of the two groups coming together and attempting to merge the two ends, fixing up issues, and adding some additional functionality that might be missing.

The fourth and final phase of the project, when members would test out the application, for its functionality and coherence, with final attempts at debugging. As the product came to a state where everyone was satisfied, a demonstration would be recorded, and a report would be written up as part of the requirements.